MATHsAiD: Automated Mathematical Theory Exploration

Appl Intell
DOI 10.1007/s10489-017-0954-8

MATHsAiD: Automated mathematical theory exploration

R. L. McCasland[1] · A. Bundy[1] · P. F. Smith[2]

Abstract The aim of the MATHsAiD project is to build a tool for automated theorem-discovery; to design and build a tool to automatically conjecture and prove theorems (lemmas, corollaries, etc.) from a set of user-supplied axioms and definitions. No other input is required. This tool would, for instance, allow a mathematician to try several versions of a particular definition, and in a relatively small amount of time, be able to see some of the consequences, in terms of the resulting theorems, of each version. Moreover, the automatically discovered theorems could perhaps help the users to discover and prove further theorems for themselves. The tool could also easily be used by educators (to generate exercise sets, for instance) and by students as well. In a similar fashion, it might also prove useful in enabling automated theorem provers to dispatch many of the more difficult proof obligations arising in software verification, by automatically generating lemmas which are needed by the prover, in order to finish these proofs.

Keywords Theory exploration · Automated theorem proving

✉ A. Bundy
A.Bundy@ed.ac.uk

R.L. McCasland
rmccasla@staffmail.ed.ac.uk

P.F. Smith
Patrick.Smith@glasgow.ac.uk

[1] School of Informatics, University of Edinburgh, Edinburgh, UK

[2] Department of Mathematics, University of Glasgow, Glasgow, UK

1 Introduction

MATHsAiD (Mechanically Ascertaining Theorems from Hypotheses, Axioms and Definitions) is a tool for assisting the working mathematician explore new mathematical theories. Given the axioms of a theory, MATHsAiD will derive from them the kind of routine theorems that would be of interest to the mathematician, either confirming that the initial axiomatisation met the original intentions or revealing some undesired consequence, e.g., proving a theorem that wasn't intended or failing to prove one that was intended. Since mathematical opinions understandably differ about which theorems are interesting, MATHsAiD is inherently *incomplete*, i.e., it will sometimes fail to derive theorems that some mathematicians *would* regard as interesting. Moreover, since it encompasses proof by induction, it is inherently incomplete as a consequence of Gödel's Incompleteness Theorems. We claim only that the difference between MATHsAiD's assessment of interestingness and that of a typical mathematician is no bigger than that between two typical mathematicians. See Section 4 for a discussion of this claim. The theorems routinely generated by MATHsAiD can then be used as lemmas in more significant theorems whose proofs exceed MATHsAiD's unaided abilities.

Note that MATHsAiD is a theorem *discovery* system, i.e., given a theory it tries to discover and prove interesting theorems in that theory. It is not a theorem prover, i.e., it was not principally designed to prove theorems provided by the user, although that is an optional mode. We call these two modes: *discovery mode* and *theorem-proving mode*. The focus of this paper is on discovery mode.

Throughout the MATHsAiD research programme we have followed the following basic principles:

Published online: 23 June 2017

Transparency: In developing a mathematical theory, MATHsAiD should use only information (axioms, definitions, rules, etc.) provided by the user or derived from this information.

Small Steps: MATHsAiD should develop a mathematical theory in rather small steps, rather than attempting any major theorems straight away. This way, at each stage only 'simple' mathematics need be done; and yet, by this approach some very sophisticated mathematics can be achieved. It is just that, by the time the system gets to the high-level maths, its database (hopefully) contains enough results so that what would otherwise be difficult, is, in fact relatively simple.

Multiple Theories: MATHsAiD must be able to deal with multiple theories. The user should be able to work in any theory in the database, and not have to tell the system which other theories are required as prerequisites for the 'working' theory.

New and Simple: Every result which is recorded by MATHsAiD and presented to the user, should be, in some sense at least, both 'new' and 'relatively simple' (as in, simpler than 'comparable' statements). The understanding and implementation of these two criteria are each rather open to interpretation. (MATHsAiD versions have already had 2 or 3 different implementations of these criteria.)

Facts *vs* Theorems: MATHsAiD might need to record some results (theorems), purely for the sake of efficiency, which do not meet all the above criteria. It is expected that such results would likely not be of much mathematical interest to users, but they might need/want access to them, all the same. MATHsAiD should make plain to the user which results do, and do not meet the above criteria.

Human-Like: In nearly every aspect of MATHsAiD's implementation, our rule of thumb is to seek a mathematical parallel as practiced by at least some research mathematicians. If a parallel can be found, good; otherwise, a change in the implementation is probably advisable.

Standard Notation: Considerable effort should be made to accommodate 'standard' mathematical notation, as far as what the user sees. It is likely that the user will not particularly care what notation MATHsAiD uses internally. So, it needs to be able to convert from this internal notation to 'user' notation. If it can convert in the other direction as well, so much the better.

Multiple Strategies: The discovery process should not be tied to a monolithic reasoning strategy, but should combine different strategies, e.g., forward and backward, in an opportunistic way.

Throughout this paper, we highlight with a footnote when we have realised one of these principles.

To be useful to working mathematicians, it is essential that MATHsAiD should be capable of conjecturing and proving theorems in theories of current mathematical interest, ideally including non-trivial theorems. In particular, it should be able to conjecture and prove theorems inter-relating different mathematical theories. These goals constitute the main aim of MATHsAiD 2.0, the current version of MATHsAiD, which is described in this paper.

The aim of the MATHsAiD 2.0 system, therefore, is:

> *To be a useful aid to the working mathematician, by conjecturing and proving many of the interesting theorems of a given mathematical theory (from user-provided axioms), whilst limiting the number of non-interesting theorems generated.*

In order to test whether MATHsAiD 2.0 could meet its aim, we set it the task of conjecturing and proving at least one recently published theorem. We targeted the theory of Zariski spaces, which was discovered and explored by the first and third authors. In particular, we hoped that MATHsAiD 2.0 could re-discover some general results about prime submodules and, more specifically, some results in the theory of Zariski spaces.

MATHsAiD 2.0 classifies the conjectures it proves as either *facts, lemmas, Theorems* and *inter-theory results*.

Facts: are intermediate results of no intrinsic mathematical interest[1]. Facts are so classified because they are trivial consequences of previously known Theorems or because they are unnecessarily complex, i.e., that they could be simplified. Each Fact is generated as a side product of the generation of a particular Theorem and is deemed to be useful only during that process. Once the Theorem generator has completed its immediate task, all the Facts that it has generated are deleted.

Lemmas: are generated by the theorem generating process but fail to meet all the criteria demanded of a Theorem. They are stored permanently in case they find a use as intermediate lemmas in the proof of subsequent Theorems, but are not reported to the user as Theorems.

Theorems: are generated by the theorem generating process and meet all the criteria demanded of a Theorem. They are stored permanently and reported to the user as Theorems.

Inter-Theory Results: are theorems that relate two different operators in two different theories[2]. These results can be facts, lemmas or Theorems.

The key filters used to determine Theoremhood are listed in Section 3.

[1] In this and the next two bullets, we realise the Facts *vs* Theorems principle.

[2] Realising the Multiple Theories principle.

Given a set of axioms and definitions for a theory, MATHsAiD 2.0 analyzes the information supplied, and based on this analysis, generates a sequence of sets of hypotheses and terms of interest. This sequence is designed to discover the more 'routine' Theorems in this theory; i.e., results that one might expect to see in an introductory mathematics textbook. For example, in set theory, given the usual definitions of union and intersection, MATHsAiD 2.0 discovers, among other things, that these operations are commutative, associative, and each is distributive over the other. For each set of hypotheses in the aforementioned sequence, MATHsAiD 2.0 uses a combination of 'generating' and 'trivial' proof plans to discover all the more-or-less interesting results it can, subject to numerous constraints. In particular, the 'generating' proof plans serve to derive (generate and prove) various conclusions c from the given hypotheses, whereas the 'trivial' proof plans act as a constraint, by allowing the assertion of a newly-derived c only if it fails to be proven by any of the 'trivial' proof plans. Once the system is no longer able to assert any additional conclusions, either because the combination of 'generating' and 'trivial' proof plans do not allow such, or because the time limit has expired, the generated conclusions are then passed to a final filtering stage, in which the less interesting ones are weeded out, leaving (hopefully) only the sorts of results that the user desires to see.

In this paper, we will use the following conventions:

- We will use 'theorem' to describe all provable formulae, but 'Theorem' to distinguish those theorems that pass the Theorem-hood criteria of Section 2, so are considered sufficiently interesting to be reported as such to the user.
- We will use the lower case letters x, y, z, possibly subscripted, to stand for meta-level variables, which we will also call *holes*.
- We will use Greek letters, possibly sub-scripted, to stand for all other meta-level expressions in patterns, e.g., to express the structure of Theorem-producing rules, the conjecture shell, term of interest schemas, etc. Occasionally, we will also use special symbols, such as $\approx$, to stand for particular kinds of meta-level expressions, in this case equivalence relations.
- An *operator* is a non-nullary function or predicate. A *constant* is a nullary function or predicate.

2 An overview of MATHsAiD 2.0

All of the theorems proved by MATHsAiD 2.0 are derivations in a logical theory, whose rules, definitions and axioms are accessible to the user[3].

[3]Realising the Transparency principle.

The logic used by MATHsAiD 2.0 is user defined, but its default logic is a classical, untyped higher-order logic. The default logic was chosen to reflect standard mathematical practice[4] and is based on Gentzen's Natural Deduction. Table 1 lists the logical rules of inference used by MATHsAiD 2.0.

Mathematicians typically do not assign types to objects, but they do use sets. We have followed the Gödel-Bernays approach to sets and classes, as found in [6], for instance, where classes are comprised of sets and proper classes[5]. Proper classes are, roughly speaking, too large to be sets, e.g., the class of all sets. Inconsistencies, such as Russell's Paradox, can arise from the application of set comprehension to classes; this dilemma is avoided by restricting comprehension to apply only to sets.

Implicit typing information is represented by unary properties, such as $Class(S)$, or binary set member relations, such as $n \in \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. These typing propositions are internally tagged by MATHsAiD 2.0, which distinguishes between type and non-type propositions during its formation of Theorems. Note that one type can be a subtype of another. For instance, *Set*s are a subtype of *Class*es. When two types share a common subtype, we will say they are *compatible*.

Each MATHsAiD 2.0's Theorem consists of a, possibly empty, hypothesis followed by a conclusion. The hypothesis is a conjunction of some type propositions and possibly some non-type propositions. Each variable in the Theorem is implicitly universally[6] quantified and its type is declared in one of these type propositions. Each conclusion contains a *term of interest*, i.e., a term that has some intrinsic mathematical interest.

MATHsAiD 2.0's proofs consist of logical and transitive reasoning, plus induction, when a theory contains inductive rules. By *transitive reasoning* we mean that a proof of $\xi_1 \ \Xi \ \xi_n$, say, takes the form:

$$\xi_1 \ \Xi \ \xi_2 \ \Xi \ \ldots \ \Xi \ \xi_n$$

where Ξ is a transitive relation, whose transitivity is invoked to then conclude that $\xi_1 \ \Xi \ \xi_n$. This is essentially rewriting.

The proofs of Theorems are quite short, i.e., a few rule applications[7]. This ensures that interesting Theorems are not overlooked by the generation process and reflects the high density of interesting Theorems in MATHsAiD 2.0's search spaces. Theorems requiring long proofs are discov-

[4]Realising the Human-Like principle.

[5]Also realising the Human-Like principle.

[6]This is a common restriction in automated provers, but this is not an inherent limitation of the approach. Work on existential quantification is currently in progress.

[7]Realising the Small Steps principle.

Table 1 The Rules of Natural Deduction: All type inheritance rules and the type antecedents on all rules have been omitted to reduce clutter. The type conventions are that: P, Q and R range over propositions, A and C over open sentences, t over terms and x and y over variables. The ys in rules $\forall I$, $\exists E$, $\forall : I$ and $\exists : E$ are fresh variables that do not occur in the $A(x)$s

Introduction Rules	Elimination Rules
$\frac{}{P \vee \neg P}\, ExMid$	$\frac{\bot}{P}\bot E$
$\frac{\begin{array}{c}P\\ \vdots\\ \bot\end{array}}{\neg P}\neg I$	$\frac{P \quad \neg P}{\bot}\neg E$
$\frac{P \quad Q}{P \wedge Q}\wedge I$	$\frac{P \wedge Q}{P}\wedge E_L \qquad \frac{P \wedge Q}{Q}\wedge E_R$
	$\frac{P \vee Q \quad \begin{array}{c}P\\ \vdots\end{array} R \quad \begin{array}{c}P\\ \vdots\end{array} R}{r}\vee E$
$\frac{P}{P \vee Q}\vee I_L \qquad \frac{Q}{P \vee Q}\vee I_R$	
$\frac{\begin{array}{c}P\\ \vdots\\ Q\end{array}}{P \implies Q}\implies I$	$\frac{P \implies Q \quad P}{Q}\implies E$
$\frac{P \implies Q \quad Q \implies P}{P \iff Q}\iff I$	$\frac{p \iff Q}{(P \implies Q) \wedge (Q \implies P)}\iff E$
$\frac{A(y)}{\forall x.A(x)}\forall I \qquad \frac{\begin{array}{c}C_{(y)}\\ \vdots\\ A_{(y)}\end{array}}{\forall x : C(x).A(x)}\forall : I$	$\frac{\forall x.A(x)}{A(t)}\forall E \qquad \frac{\forall x : C(x).A(x) \quad C(t)}{A(t)}\forall : E$
$\frac{A(t)}{\exists x.A(x)}\exists I \qquad \frac{A(t) \quad C(t)}{\exists x : C(x).A.(x)}\exists : I$	$\frac{\exists x.A(x) \quad \begin{array}{c}A_{(y)}\\ \vdots\\ P\end{array}}{P}\exists E \qquad \frac{\exists x : C(x).A(x) \quad \begin{array}{c}A_{(y)} \quad C_{(y)}\\ \vdots\\ P\end{array}}{P}\exists : E$

ered by the accumulation and combination of intermediate lemmas, each of which is a theorem.

Theorems are generated in parallel with their proof, which is partly by a forwards reasoning process. First, the hypothesis, the conclusion's main predicate and a term of interest are generated. These are combined to form a Theorem shell, i.e., some *holes* in the Theorem remain unspecified. Then a forward inference process explores what conclusions containing this term of interest follow from the hypothesis. The holes in the Theorem are filled in as a side-effect of its proof.

The proof process is guided by *sketch plans*, which we have abstracted from common patterns of reasoning observed in human proofs[8] These help to ensure that interesting Theorems are generated. There are two kinds of sketch plans: generating and trivial. Generating plans are used to generate interesting Theorems and trivial plans to check that they are not merely facts, i.e., if a formula can be proved by a trivial plan, then it is classified as a fact rather than a Theorem[9]. In addition, the last step of the proof of a Theorem must be by a *Theorem-producing rule*. These are rules whose conclusions are no more complex than their hypotheses, i.e., a rule of the form:

$$\theta_1 \wedge \ldots \wedge \theta_n \implies \gamma$$

where:

$$size(\gamma) \leq max_{i \in [n]} size(\theta_i)$$

and $size$ counts the number of operators in an expression, as a measure of its complexity. At least two of the θ_i must be non-type propositions. The restriction to these rules helps ensure the simplicity of the Theorems.

[8]Realising the Human-Like principle.

[9]Realising the New and Simple principle.

In some cases one cannot check that a candidate Theorem-producing rule meets the non-increasing size criterion until it is known how it will be instantiated within a proof. Consider, for instance, a transitivity rule, such as:

$$x = y \wedge y = z \Longrightarrow x = z$$

MATHsAiD 2.0 cannot tell whether:

$$size(x = z) \leq max(size(x = y), size(y = z))$$

until it is known what y will be instantiated to. This problem occurs whenever a variable occurs in a θ_i that does not occur in γ. Such rules are called *conditional* Theorem-producing rules and the non-increasing size criterion is checked dynamically once a rule has been fully instantiated in use.

There are a few Theorems that cannot be proved using terminal applications of Theorem-producing rules but are too important to exclude, e.g., the transitivity of $\Longrightarrow$. Its standard proof is:

$$\dfrac{\dfrac{P, \quad P \Longrightarrow Q}{Q,} \qquad Q \Longrightarrow R}{\dfrac{R}{P \Longrightarrow R}}$$

but in this proof the final implication, $Q \Longrightarrow R$, fails to be a Theorem producing rule because it only has one non-type antecedent. In fact, none of the implications used in the proof meets this requirement, ruling out the possibility of some meeting the requirement by some rearrangement of the proof. We could try to introduce a redundant Theorem-producing rule in order to meet the requirement, but this would not just be ugly but would probably exceed our maximum proof length and not be discovered.

Such Theorems are instead generated by instantiating schemas, e.g., of transitivity, and are proved by backward reasoning from these instantiated schemas[10].

3 Implementation

In outline, the Theorem/lemma generation process consists of three stages:

1. A conjecture shell is constructed using the material provided by the hypothesis and term of interest generators described in Section 3.1. This material consists of a hypothesis θ, a term of interest ξ and a k-ary predicate, ρ. The conjecture shell is then:

   $$\theta \Longrightarrow \rho(y_1, \ldots, \xi, \ldots, y_k) \qquad (1)$$

   where the y_i are distinct new variables called *holes*.

2. The sketch plans are used to reason backwards from this conjecture shell. If successful, this backwards reasoning both generates a proof sketch and instantiates the y_i, i.e., fills in the holes. Note that this process usually fails, either because the conjecture shell cannot be instantiated to a true formula or because MATHsAiD fails to outline a proof of it. If it succeeds then some filters are applied to identify and discard mere facts.

   (a) The proof sketch must outline a non-trivial proof.
   (b) The last rule applied in the proof sketch must be a Theorem producing rule.
   (c) The conjecture must not also have a trivial proof. An attempt is made to prove it using some very simple techniques, such as showing that it is an instance of an existing Theorem.

   Only conjectures that pass all these filters proceed to the final stage.
3. The fully instantiated conjecture and its proof sketch are sent to the theorem prover, which uses a process of forwards reasoning to turn the proof sketch into a full proof. If this succeeds then the conjecture is reclassified as a Theorem.

Not all these new theorems are equally interesting. Some are useful as lemmas to use in subsequent proofs, so need to be stored for reuse, but are not intrinsically interesting in their own right. If ρ is instantiated to a type predicate, then the result is a lemma. On the other hand, if the predicate ρ in the conjecture shell (1) is instantiated to = during its proof, then the result is usually a Theorem. An exception to this is *two-results*, as explained in Section 3.4.7.

Unfortunately, not all Theorems and lemmas can be produced by the process outlined above and in Section 3.1 below. For instance, the Theorem-producing rule filter ((2b) above) excludes some standard Theorems, e.g., transitivity Theorems. Also, some Theorems are produced in a non-standard format, e.g., monotonicity Theorems. So an additional Theorem generation mechanism is also used. Schemas describing the shape of these lemmas and Theorems are instantiated and then an attempt is made to prove them. This additional mechanism and the lemmas and Theorems it produces is described in Section 3.2. Note that all lemmas are produced top-down from schemas.

3.1 Hypothesis and term of interest generation

The hypothesis generator and the term of interest generator identify a predicate, ρ, and use it to generate the term of interest, ξ, the hypothesis, θ, and the conjecture shell (1).

Each predicate in the current theory is a candidate for ρ. Each candidate is used in combination with each compatible candidate for θ and ξ.

[10] Realising the Multiple Strategies principle.

3.1.1 Term of interest generation

The term of interest generator works by systematically and exhaustively instantiating a set of schemas up to some resource limits, namely term size and nesting bounds. Each possible instantiation of a schema is a possible candidate for the term of interest ξ. For instance, the schemas used for a binary operator μ are:

- $\mu(x, y)$ in all situations.
- $\mu(x, x)$ if the arguments of μ have compatible types.
- $\mu(\mu(x, y), z)$ and/or $\mu(x, \mu(y, z))$ if the result of μ has a type compatible with one or both of its arguments.
- $\mu(c, x)$ and/or $\mu(x, c)$ if there is also a constant c with a type compatible with one or both of μ's arguments.
- $\nu(\mu(x, y))$, $\mu(\nu(x), y)$, $\mu(x, \nu(y))$, $\mu(\nu(x), \nu(y))$ if there is also another unary operator ν, where the type of μ's result is compatible with the type of ν's argument or vice versa.
- $\nu(\mu(x, y), z)$, $\nu(x, \mu(y, z))$ and/or $\nu(\mu(x, y), \mu(z, w))$ if there is also another binary operator ν, where the type of μ's result is compatible with one or both of the types of ν's arguments. And ditto with the roles of μ and ν reversed.

A term of interest is used to generate zero, one or more Theorems. If one is generated that has already been used in an equivalence relation in an existing Theorem, then it is filtered out.

3.1.2 Hypothesis generation

The hypotheses of each conjecture consist of two parts: *type declarations* and *non-type hypotheses*.

The type declarations can be easily calculated from the term of interest ξ. For each variable, say x, in ξ, we must first identify its type, say τ, and then create a hypothesis asserting that type, say $\tau(x)$. The identification of the type τ must take into account any compatibility conditions accumulated during the construction of the term of interest. For instance, suppose the term of interest is $\mu(x, x)$, where μ has type $\tau_1 \times \tau_2 \mapsto \tau_3$. To form this term of interest τ_1 and τ_2 must be compatible, say τ_1 is a subtype of τ_2. Now the type of x is restricted to τ_1. The result of this stage of hypothesis generation will be a conjunction of type declarations of the form $\tau_1(x_1) \wedge \ldots \wedge \tau_n(x_n)$.

A non-type hypothesis is generated for each predicate ς in the theory that is different from ρ and for each combination of arguments chosen from compatible x_i. Suppose, for instance, that ς has type $\tau'_1 \times \ldots \times \tau'_m \mapsto \tau'$. A hypothesis of the form $\varsigma(x'_1, \ldots, x'_m)$ is formed for each combination of $x'_1, \ldots, x'_m$ where the type of x'_i is compatible with τ'_i and each x'_i is x_j for some j.

A complete hypothesis, θ, now consists of all the type declarations plus zero or more non-type hypothesis.

3.1.3 Lemma generation

Lemmas are formed from conjectures of the form (1) by exhaustive execution of the following procedure:

Procedure 1 (Lemma Generation)

1. *Pick a k-ary predicate* ρ.
2. *Pick an argument of* ρ *and generate a term of interest* ξ *whose type is compatible with that argument of* ρ. *Fill the rest of the arguments of* ρ *with distinct variables* y_i.
3. *Use the term of interest to form a hypothesis* θ.
4. *Form the conjecture:*

$$\theta \Longrightarrow \rho(y_1, \ldots, \xi, \ldots, y_k)$$

Example 1 **(Lemma Generation)**
Consider the following lemma:

$$Class(A) \wedge Class(B) \wedge \quad Class(C) \wedge R : A \mapsto B \wedge S : B \mapsto C \Longrightarrow S \oplus R : A \mapsto C \tag{2}$$

where $S \oplus R ::= \lambda x.\, S(R(x))$, i.e., the composure of S and R, and $R : A \mapsto B$ means R is a function from A to B.

The above procedure generates this lemma as follows:

- Let ρ be the ternary predicate $\ldots : \ldots \mapsto \ldots$, which is not an equivalence relation.
- Pick the first argument of ρ and generate the term of interest ξ to be $S \oplus R$.
- One of the hypotheses generated is:

$$Class(A) \wedge Class(B) \wedge Class(C) \wedge R : A \mapsto B \wedge S : B \mapsto C$$

- The conjecture shell has now been instantiated to (2).

A common kind of lemmas discovered by MATHsAiD 2.0 are type inheritance rules, e.g.,

$$Class(A) \wedge Class(B) \Longrightarrow Class(A \cap B)$$

Type inheritance conjectures are generated for every operator. For instance, suppose the arguments of a binary operator μ are both of type τ, but its output type is not known. The following schema is used:

$$\tau(x) \wedge \tau(y) \Longrightarrow \tau(\mu(x, y))$$

to try to prove that the output type is also τ.

A special case of lemma production is *two-results*. These are formed from terms of interest containing recursively defined operators and are used in the construction of inductive conjectures. Suppose ξ is a term of interest containing a recursively defined operator and a universally quantified variable, say x of type τ. MATHsAiD 2.0 calculates a *two-object* for terms of type τ, say c, and substitutes c for x in

ξ to form a *two-term*, say ξ'. *Two-objects* are terms formed from two applications of a step constructor to a base constructor, e.g., $s(s(0))$, $cons(a, cons(b, nil))$, etc., where s is the successor function, i.e., $s(0)$ represents 1, $s(s(0))$ represents 2, etc., and *cons* is the function to attach a new element to the head of a list. Suppose, for instance, that the term of interest is $m+n$, where $+$ is recursively defined and m and n are universally quantified and have type $\mathbb{N}$. The two-object for type $\mathbb{N}$ is s(s(0)), and $s(s(0)) + n$ and $m + s(s(0))$ are the two-terms formed from $m + n$.

The two-terms are then adopted as new terms of interest to create *two-results*, i.e., theorems in which these two-cases form the left-hand side of an equation. In our example, a process of forwards reasoning from the two-terms produces the following two-results:

$$\forall n \in \mathbb{N} \Longrightarrow s(s(0)) + n = s(s(n)) \tag{3}$$

$$\forall m \in \mathbb{N} \Longrightarrow m + s(s(0)) = s(s(m)) \tag{4}$$

These two-results are then proved, which results in (4) being rejected as trivial but (4) being used to suggest an inductive conjecture. It is re-generalised by replacing the two-objects with universal variables to form the following inductive conjecture:

$$m + n = n + m \tag{5}$$

where each $s(s(0))$ has been replaced by m.

A sketch plan for each inductive conjectures is sought. If its proof sketch is trivial then it is rejected. If not, and a full inductive proof is found, the conjecture is adopted as a Theorem. Conjecture (5) is the commutativity of $+$, which *does* have an interesting sketch plan and which *can* be proved by induction. More details of this process can be found in [14].

3.1.4 Theorem generation

Theorems are formed from conjectures of the form:

$$\theta \Longrightarrow \rho(y_1, \ldots, \xi, \ldots, y_k)$$

by exhaustive execution of the following procedure.

The term of interest and hypothesis generators instantiate ξ and θ in the above conjecture shell. This partially instantiated shell is now passed to the *theorem generator*. Its role is to instantiate ρ and prove the resulting Theorem. It uses the following procedure.

Procedure 2 (Theorem Generation)

1. *Non-deterministically choose a Theorem-producing rule:*

   $$\theta_1 \wedge \ldots \wedge \theta_n \Longrightarrow \gamma \tag{6}$$

   such that $\rho(y_1, \ldots, \xi, \ldots, y_k)\sigma_2 \equiv \gamma\sigma_1$ *for some substitutions* σ_1 *and* σ_2. *The search for suitable* (6), σ_1 *and* σ_2 *proceeds as follows:*

   For each candidate (6)*:*

   (a) *Suppose* θ_j *is the first hypothesis of maximum size, i.e.,* $size(\theta_j) = max_{i\in[n]} size(\theta_i)$.
   (b) *If possible, instantiate* θ_j *so that it contains* ξ *as a subterm, i.e.,* ξ *occurs in* $\theta_j\sigma_1$ *for some substitution* σ_1, *otherwise terminate with failure.*
   (c) *Now match* $\rho(y_1, \ldots, \xi, \ldots, y_k)$ *to* $\gamma\sigma_1$ *with substitution* σ_2, *otherwise terminate with failure.*

   Note that ρ *is now fully instantiated and that the term of interest* ξ *occurs in both the hypothesis and the conclusion of the Theorem-producing rule. The first step of backwards reasoning has also been accomplished and the new sub-goal is:*

   $$\theta\sigma_2 \Longrightarrow (\theta_1 \wedge \ldots \wedge \theta_n)\sigma_1 \tag{7}$$

3. *Use the sketch plans to generate a proof sketch for* (7) *by backwards reasoning. The proof sketch is assessed. If it is deemed too trivial to justify classifying the conjecture as a Theorem, then terminate with failure.*
1. *Execute the proof sketch by forwards reasoning to produce a full proof.*

This procedure is repeated for all Theorem-producing rules (6) *and all ways to fit* ξ *to* θ_j.

Example 2 **(Theorem Generation)**

For instance, suppose that the term of interest ξ is $A \cap B$, so that the conjecture shell is:

$$Class(A) \wedge Class(B) \Longrightarrow P(y_1, \ldots, A \cap B, \ldots, y_k)$$

and the transitivity of = is selected as the Theorem-producing rule:

$$x = y \wedge y = z \Longrightarrow x = z \tag{8}$$

where $x = y$ is θ_j.

– One instantiation of θ_j to contain ξ is to make the substitution σ_1 be $\{A \cap B/x\}$ where n is 2, so that (8) is:

  $$A \cap B = y \wedge y = z \Longrightarrow A \cap B = z$$

– Matching $\rho(y_1, \ldots, A \cap B, \ldots, y_k)$ to $A \cap B = z$ makes the substitution σ_2 be $\{z/y_1, = /\rho\}$ where k is 2.
– The sketch plans are now used to instantiate and prove:

  $$Class(A) \wedge Class(B) \Longrightarrow A \cap B = y \wedge y = z$$

  by backwards reasoning. During this process z is instantiated to $B \cap A$ and y to $\{w : w \in A \wedge w \in B\}$.
– This instantiates the conjecture to:

  $$Class(A) \wedge Class(B) \Longrightarrow A \cap B = \{w : w \in A \wedge w \in B\} \wedge \{w : w \in A \wedge w \in B\} = B \cap A$$

– The transitive chain can now be collapsed to give the final conjecture as:

$$Class(A) \wedge Class(B) \Longrightarrow A \cap B = B \cap A$$

The sketch plan is then used to generate the full proof of this conjecture by forwards reasoning.

Some other examples of Theorems proved by MATHsAiD 2.0, labelled by the theories in which they are proved, are:

Classes:

$$Class(A) \wedge Class(B) \wedge Class(C) \Longrightarrow A \setminus (B \setminus C) = (A \setminus B) \cup (A \cap C)$$

Zariski Topology:

$$comRingWOne(R) \Longrightarrow \zeta(R)\ topologyOn\ spec(R)$$

Zariski Spaces:

$$comRingWOne(R) \wedge leftUnitalModule(M, R) \Longrightarrow leftSemimodule(\zeta(M), \zeta(R))$$

3.2 Lemma and theorem schemas

As mentioned in Section 3, to complement the theorem generation process described in Section 3.1, MATHsAiD 2.0 also generates lemmas and Theorems by the instantiation of schemas. The schemas are divided into two types: those producing lemmas and those producing Theorems. Below we describe these two kinds of schema and give examples of the kind of lemmas and Theorems they produce.

3.2.1 Lemma schemas

The following schemas are used to produce positive or negative monotonicity rules, which are classified as lemmas. These lemmas show how a relationship is inherited or inverted under operator application, e.g.,

$$Class(A) \wedge Class(B) \wedge Class(R) \Longrightarrow (A \subset B \Longrightarrow A \cup R \subset B \cup R)$$

Monotonicity conjectures are generated for every combination of transitive, binary predicates Ξ_i (of type $\tau \times \tau \mapsto \tau$) and unary and binary operators μ_1 (of type $\tau \mapsto \tau$) and μ_2 (of type $\tau \times \tau \mapsto \tau$) using the schemas:

$$\tau(x) \wedge \tau(y) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_1(x)\ \Xi_2\ \mu_1(y))$$
$$\tau(x) \wedge \tau(y) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_1(y)\ \Xi_2\ \mu_1(x))$$
$$\tau(x) \wedge \tau(y) \wedge \tau(z) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_2(x, z)\ \Xi_2\ \mu_2(y, z))$$
$$\tau(x) \wedge \tau(y) \wedge \tau(z) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_2(z, x)\ \Xi_2\ \mu_2(z, y))$$
$$\tau(x) \wedge \tau(y) \wedge \tau(z) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_2(y, z)\ \Xi_2\ \mu_2(x, z))$$
$$\tau(x) \wedge \tau(y) \wedge \tau(z) \Longrightarrow (x\ \Xi_1\ y \Longrightarrow \mu_2(z, y)\ \Xi_2\ \mu_2(z, x))$$

where Ξ_1 and Ξ_2 may or may not be the same predicate.

3.2.2 Theorem schemas

The following schemas are used to produce Theorems, i.e., results that *are* intrinsically interesting.

Reflexivity rules: These show the reflexivity of binary predicates, e.g.,

$$Prop(P) \Longrightarrow (P \Longrightarrow P)$$

Reflexivity conjectures are generated for every binary predicate ϕ (of type $\tau \times \tau \mapsto bool$) using the schema:

$$\tau(x) \Longrightarrow \phi(x, x)$$

Transitivity rules: These show the transitivity of binary predicates, e.g.,

$$Class(A) \wedge Class(B) \wedge Class(C) \Longrightarrow (A \subset B \wedge B \subset C \Longrightarrow A \subset C)$$

Transitivity conjectures are generated for every binary predicate ϕ (of type $\tau \times \tau \mapsto bool$) using the schema:

$$\tau(x) \wedge \tau(y) \wedge \tau(z) \Longrightarrow (\phi(x, y) \wedge \phi(y, z) \Longrightarrow \phi(x, z)$$

Quantifier distributivity rules: These distribute quantifiers over connectives, e.g.,

$$\forall z.\ (P(z) \wedge Q(z)) \Longrightarrow (\forall x.;\ P(x) \wedge \forall y.;\ Q(y))$$

and transitive relations, e.g.,

$$\forall z.\ (P(z) \iff Q(z)) \Longrightarrow \{x : P(x)\} = \{y : Q(y)\}$$

Quantifier distributivity conjectures are generated for every transitive relation π (of type $\tau \times \tau \mapsto bool$) using the schema:

$$\forall z.\ (\tau(z) \Longrightarrow \Xi(\phi(z), \psi(z))) \Longrightarrow \pi(\Pi x.\ \phi(x), \Pi y.\ \psi(x))$$

where Ξ is either $\Longrightarrow$ or $\iff$ and Π is either $\forall$, $\exists$ or set comprehension, e.g., $\{x : \phi(x)\}$. Conjectures are made for all well-defined combinations.

Converses: Provided the user has selected the 'converse' option, then there is an attempt to turn all implications into equivalences, i.e., if $\theta \Longrightarrow \gamma$ has been proved, then MATHsAiD 2.0 will conjecture $\gamma \Longrightarrow \theta$.

Once a schema has been instantiated to a conjecture, then an attempt is made to prove it.

3.3 Inter-theory result generation

An inter-theory result relates two or more different operators in two or more different theories[11]. For instance, the $\subseteq$

[11] Realising the Multiple Theories principle.

operator in the theory of Sets is an example of a partial order $\preceq$ from the theory of Orderings. Such an inter-theory result is established by showing that an operator from one theory meets the definition of an operator from another theory. In our example, a partial order $\preceq$ is defined to be a binary operator that is reflexive, transitive and antisymmetric. The $\subseteq$ operator is then shown to meet these defining properties.

A more challenging example is illustrated in Fig. 2, where the Theorem marked 'Theorem (15)' shows that given an R-module M (R is a commutative ring with 1 and M is unital) the set $\zeta(M)$ of all varieties of subsets of M forms a semimodule over the Zariski topology of R (viewed as a semiring), with an appropriate choice of scalar multiplication. This is essentially Theorem 2 from [13]. MATHsAiD 2.0's discovery and proof of this Theorem shows both that it can reason with high-level theories and that it can conjecture and prove mathematics of current research interest.

It is important to note that derivation of Theorem 15 is only made possible because of certain inter-theory results that MATHsAiD 2.0 has inferred earlier in its theory exploration run. Admittedly, many of these inter-theory results would likely be judged, solely on their own merits, to be rather uninteresting. Nevertheless, they can be, in effect, important stepping stones[12]. We will refer to Theorems (such as Theorem 15), which have this relationship to inter-theory results, as inter-theory Theorems.

By *high-level* theory we mean a theory that is built on top of a lattice of other theories. Zariski spaces are built on the theories of topology and (semi)modules; (semi)modules are in turn built on (semi)rings, which are built on (semi)groups, etc. All of these theories are built on class/set theory, which is built on logic. This tower is illustrated in Fig. 1.

3.4 Sketch plans and proof sketches

The theorem generation process described in Section 3.1.4 uses *sketch plans* for three purposes:

- to complete the instantiation of a partially instantiated conjecture shell;
- to produce a *proof sketch* to guide the search for a proof; and
- to assess the proof sketch to ensure the result is worthy of classification as a Theorem.

Except for the sketch plan *forwards reasoning*, the sketch plans work backwards from the conjecture to the axioms and previously proved lemmas and Theorems[13]. To keep proofs short, there is a user-defined limit on the number of times each sketch plan can be successively applied, e.g., twice. The sketch plans pass information between them on: which sketch plans should be called next; limits on the number of further applications of a sketch plan; whether or not a sketch plan must fully instantiate the conjecture; whether the conjecture will yield a lemma or a Theorem; etc.

A proof sketch falls short of a full proof mainly in that no type checking is done, so that some formulae may not be well formed. Additionally, some parts of the proof may be omitted with indicators of what must be done to complete it, e.g., use another part of the proof as a guide, fill in missing steps in a transitive chain, etc.

A sketch plan is an AND tree in which the nodes are sub-goals and the arcs are instances of rules of inference. Indicators are attached to nodes to advise the theorem proving procedure when the sketch is completed. These indicators are:

Derivation Needed: *This is a non-leaf sub-goal so the proof sketch below it needs to be unpacked.* This indicator is added by the Derivation sketch plan.

Symmetry: *The proof of this sub-goal is similar to one that has already been proved and whose proof can be used as a guide.* This indicator is added whenever symmetry is used by the sketch plans of Replacement, Simplification, Transitivity handler, Induction or Targeted Manipulation.

Previously Proved: *This sub-goal has previously been proved elsewhere, so does not need to be reproved.* This indicator may be added by the Derivation and Targeted manipulation sketch plans.

Transitive chain completion: *The proof of this sub-goal involves a transitive chain that must be unpacked*, as only the first and last sub-goal of the chain, plus the transitive relation used, are present in the proof sketch. This indicator may be added by the Simplification, Transitivity handler, Induction and Targeted Manipulation sketch plans.

The sketch plan application is controlled by a *cascade* process, which successively applies them until no sub-goals remain. To avoid duplicated effort, the cascade first checks that a sub-goal has not been previously asserted, proved or sketched, before applying sketch plans to it.

Below we outline each of the sketch plans used by MATHsAiD 2.0.

3.4.1 Derivation

Consider the $\Longrightarrow I$ rule from Table 1.

$$\frac{\begin{array}{c}P\\ \vdots \\ Q\end{array}}{P \Longrightarrow Q} \Longrightarrow I$$

[12] Realising the Small Steps principle.

[13] Realising the Multiple Strategies principle.

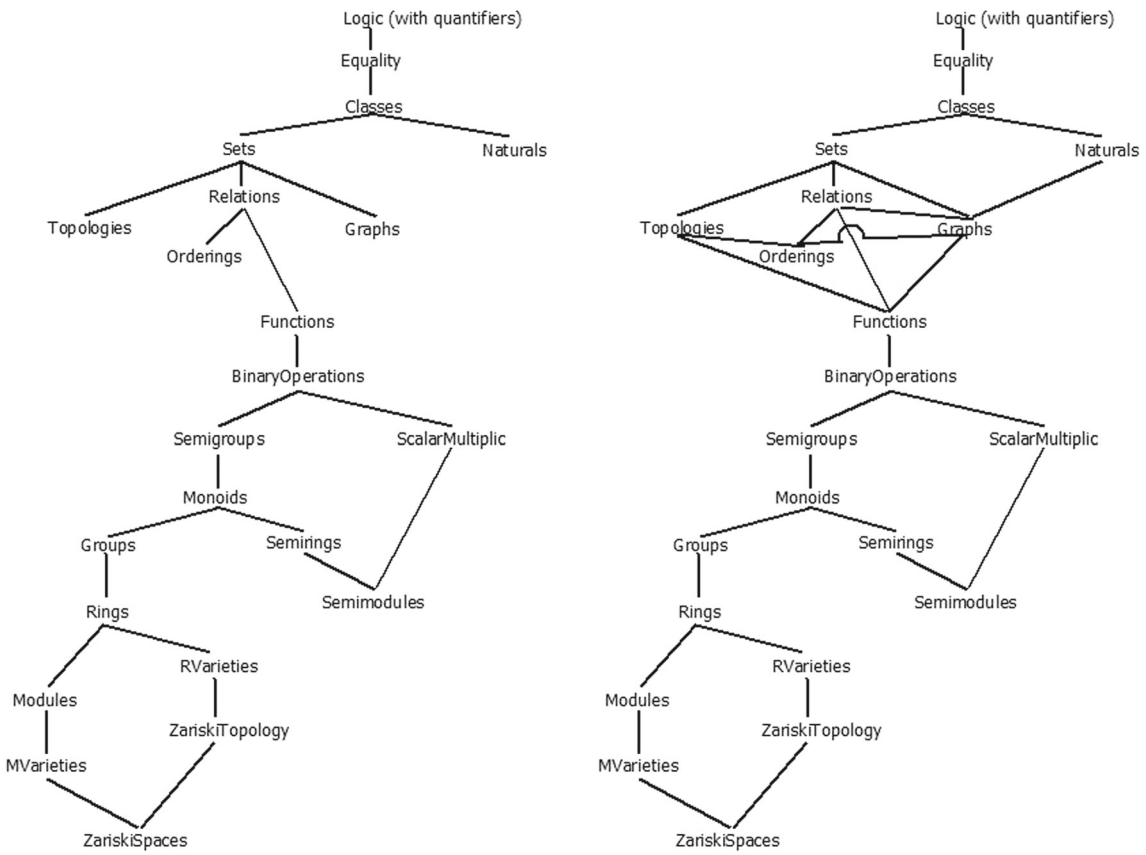

Fig. 1 Lattices of Theories Developed by MATHsAiD 2.0. Each node in these two lattices is labelled by a logical theory. Arcs between the nodes indicate a relationship between their corresponding theories. The lattice on the left shows the initial lattice of a typical run of MATHsAiD 2.0, where the loading of theories is done automatically. The one on the right shows the final state of the lattice, where the user has decided to load additional theories (beyond those loaded automatically), resulting in MATHsAiD 2.0 inferring new relationships between these and other theories. These new relationships are inter-theory results

Like five of the other logical rules in that Table, one of its antecedents is a nested derivation — in this case of Q from P. To prove such a nested derivation, MATHsAiD 2.0 makes P into a temporary assumption and then tries to prove Q from it. At the end of this attempt, whether successful or not, the temporary assumption of P is withdrawn.

The Derivation sketch plan is responsible for such nested derivations. It combines the Forwards Reasoning sketch plan from any assertions and the Backwards Reasoning sketch plan from the subgoal, trying to get them to meet in the middle. It sets a limit on the number of applications of each of these sketch plans.

Example 3 **(Derivation Sketch Plan)** Suppose the current goal is:

$$A \wedge B \Longrightarrow x$$

and that the $\Longrightarrow I$ rule has been used to set up a hypothetical context in which $A \wedge B$ is asserted. Derivation first calls the Forward Reasoning sketch plan to draw conclusions from this assumption. From this assertion, two applications of Forwards Reasoning are possible using $\wedge$ elimination.

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B}$$

i.e., both A and B are deduced.

Derivation next calls Backwards Reasoning from the goal x. The situation is as follows:

$$\frac{A, \ B}{x}$$

Two applications of Backwards Reasoning are possible using the $\wedge$ introduction rule. One of these instantiates x to $B \wedge A$ to give.

$$\frac{A, \ B}{B \wedge A}$$

Discharging the hypothetical assumption gives the Theorem:

$$A \wedge B \Longrightarrow B \wedge A$$

where the hole x has now been instantiated to $B \wedge A$.

3.4.2 Backwards reasoning

This sketch plan applies a rule backwards by unifying the current goal with the conclusion of the rule and creating new sub-goals from the instantiated hypotheses of the rule, i.e.,

$$\frac{\theta_1\sigma \wedge \ldots \wedge \theta_n\sigma}{\zeta\sigma}\theta_1 \wedge \ldots \wedge \theta_n \Longrightarrow \gamma$$

where ζ is the goal, $\theta_1 \wedge \ldots \wedge \theta_n \Longrightarrow \gamma$ is a rule with the type hypotheses elided, and $\gamma\sigma \equiv \zeta\sigma$, for some substitution σ. The $\theta_i\sigma$ become the n new sub-goals.

The following conditions must be met for backwards reasoning to apply:

- The limit on the number of successive applications of backwards reasoning must not have been reached; and
- If this rule application is required to fully instantiate ζ then $\zeta\sigma$ must contain no uninstantiated holes.

The count of successive backwards reasoning applications is now incremented and the indicator of full or partial instantiation is updated.

Example 4 **(Backwards Reasoning)**

The following inference step is by backwards reasoning:

$$\frac{(A \vee B) \Longrightarrow (B \vee A) \wedge (B \vee A) \Longrightarrow (A \vee B)}{(A \vee B) \iff (B \vee A)}$$

where the rule is:

$$(P \Longrightarrow Q \wedge Q \Longrightarrow P) \Longrightarrow (P \iff Q)$$

and σ is $\{(A \vee B)/P, (B \vee A)/Q\}$.

The original goal $(A \vee B) \iff (B \vee A$ is replaced by the two new subgoals $(A \vee B) \Longrightarrow (B \vee A)$ and $(B \vee A) \Longrightarrow (A \vee B)$

3.4.3 Forward reasoning

This sketch plan[14] applies a rule forwards by unifying a previously proved formula or current hypothesis with one of the hypotheses of the rule, proving the remaining instantiated rule hypotheses and deducing the instantiated rule conclusion, i.e.,

$$\frac{\phi\sigma \wedge \theta_2\sigma \wedge \ldots \wedge \theta_1\sigma}{\gamma\sigma}\theta_1 \wedge \theta_2 \ldots \wedge \theta_n \Longrightarrow \gamma$$

where ϕ is a previously proved formula or current hypothesis such that $\phi\sigma \equiv \theta_1\sigma$. Without loss of generality we assume that ϕ is matched to the first hypothesis of the rule.

[14]Note that the Forward Reasoning sketch plan is to be distinguished from the use of forwards reasoning to complete full proofs from proof sketches.

As with backward reasoning, we elide all type hypotheses from the rule, so that none of the dominant predicates of ϕ or the θ_i is a type predicate. Note that the $\theta_i\sigma$ must all be proved for $2 \leq i \leq n$ before $\gamma\sigma$ can be deduced.

Example 5 **(Forwards Reasoning)** The following inference step is by forwards reasoning:

$$\frac{((A \vee B) \wedge ((A \vee B) \iff (B \vee A)))}{B \vee A}(P \wedge (P \iff Q)) \Longrightarrow Q$$

where $A \vee B$ is assumed known and the hypothesis $(A \vee B) \iff (B \vee A)$ remains to be proved before the conclusion $B \vee A$ can be deduced.

3.4.4 Replacement

This sketch plan uses monotonicity lemmas to replace one subterm with another, i.e.,

$$\frac{\xi_1 \approx \xi_2}{\phi(\xi_1)\ \Xi\ \phi(\xi_2)}(x_1 \approx x_2) \Longrightarrow (\phi(x_1)\ \Xi\ \phi(x_2))$$

where $\approx$ is an equivalence relation and Ξ is a transitive relation.

Example 6 **(Replacement)**

The following inference step is by replacement:

$$\frac{(s(a+b) = s(a+c)) \iff a+b = a+c}{(s(a+b) = s(a+c)) \iff x}y = z \Longrightarrow s(y) = s(z)$$

where a, b, c are natural numbers and s is the successor function. Note how the hole x is instantiated as a side effect of the sketch plan application. Replacement could be used again to derive the goal $b = c$, which might, for instance, be an induction hypothesis.

3.4.5 Simplification

This sketch plan replaces a sub-term of the goal with an equivalent but simpler expression, i.e.,

$$\frac{\phi[\xi]\ \Xi\ \phi[\xi']}{\phi[\xi]\ \Xi\ x}\alpha \approx \beta$$

where $\alpha\sigma \equiv \xi$, $\beta\sigma \equiv \xi'$, $\approx$ is an equivalence relation and

$$(y \approx z) \Longrightarrow (\phi[y]\ \Xi\ \phi[z])$$

is a previously proved positive monotonicity lemma. Ξ is a transitive relation and $m(\xi') < m(\xi)$, where m is a measure of simplicity, e.g., the syntactic size of a term. Note that the hole x is instantiated to $\phi[\xi']$ by this sketch plan application.

Example 7 **(Simplification)**
The following inference step is by simplification:

$$\frac{(A \Longrightarrow (\neg B \vee C)) \iff (A \Longrightarrow (B \Longrightarrow C))}{(A \Longrightarrow (\neg B \vee C)) \iff x}(\neg P \vee Q)$$
$$\iff (P \Longrightarrow Q)$$

where α is $\neg P \vee Q$, β is $P \Longrightarrow Q$, $\approx$ is $\iff$, $\phi[\ldots]$ is $A \Longrightarrow \ldots$, ξ is $\neg B \vee C$, ξ' is $B \Longrightarrow C$, Ξ is $\iff$, $m(B \Longrightarrow C) < m(\neg B \vee C)$ and:

$$(y \iff z) \Longrightarrow ((A \Longrightarrow y) \iff (A \Longrightarrow z))$$

is a previously proved positive monotonicity lemma.

3.4.6 Transitivity Handler

Like simplification, this sketch plan also starts by replacing a sub-term of the goal with an equivalent expression, but it has more restrictive preconditions and does a great deal more subsequent work by rewriting the resulting goal, i.e.,

$$\frac{\phi[\xi] \ \Xi \ \phi'}{\phi[\xi] \ \Xi \ x}\alpha \approx \beta$$

where $\approx$ is an equivalence relation, Ξ is a transitive relation and $\alpha\sigma \equiv \xi$, for some substitution σ. This sketch plan is similar to simplification, but note that the right hand side of the new goal has been further rewritten to ϕ'. The preconditions of the transitivity handler sketch plan are more liberal than simplification in allowing ϕ to be a quantified expression, i.e., ϕ can be dominated by either $\forall$, $\exists$ or set comprehension.

Initially, $\alpha \approx \beta$ is used, together with the positive monotonicity lemma:

$$(y \approx z) \Longrightarrow (\phi[y] \ \Xi \ \phi[z])$$

on the goal $\phi[\xi] \ \Xi \ x$ to derive $\phi[\xi] \ \Xi \ \phi[\beta\sigma]$, which is then rewritten into $\phi[\xi] \ \Xi \ \phi'$, where ϕ' is in normal form.

Example 8 **(Transitivity Handling)**
Suppose $A \cap (B \cup C)$ is the term of interest and the following transitive chain has already been formed:

$$A \cap (B \cup C) = \{x : x \in A \wedge (x \in B \vee x \in C)\}$$

so the right hand side becomes the new goal. Note that it is dominated by set comprehension. Then an example of transitivity handling is:

$$\frac{\{x : x \in A \wedge (x \in B \vee x \in C)\}}{(A \cap B) \cup (A \cap C)}$$

using the rule:

$$\forall x : Element(x).\ P(x) \iff Q(x)$$
$$\iff \{x : P(x)\} = \{x : Q(x)\} \tag{9}$$

where $Element(x)$ means that x is an element of some set. This is a device to avoid Russell's paradox.

$P(x)$ is first matched to $x \in A \wedge (x \in B \vee x \in C)$, then the distributivity of $\wedge$ over $\vee$:

$$\forall x.\ R(x) \wedge (S(x) \vee T(x)) \iff$$
$$(R(x) \wedge S(x)) \vee ((R(x) \wedge T(x))$$

is used to instantiate $Q(x)$ to $(x \in A \wedge x \in B) \vee (x \in A \wedge x \in C)$. The whole goal is then simplified into normal form, which is

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

3.4.7 Induction

This sketch plan applies an induction rule in a recursive theory. A typical induction rule is the one for natural numbers, i.e.,

$$\frac{P(0), \quad \forall n \in \mathbb{N}.\ P(n) \Longrightarrow P(s(n))}{\forall n \in \mathbb{N}.\ P(n)} \tag{10}$$

This induction rule gives rise to one *base case*, $P(0)$, and one *step case*, $\forall n \in \mathbb{N}.\ P(n) \Longrightarrow P(s(n))$, but, in general, there could be several of each. Within the step case, $P(n)$ is called the *induction hypothesis* and $P(s(n))$ is called the *induction conclusion*. Note that all induction rules are Theorem-producing rules.

Example 9 **(Induction)**
For instance, applying the induction rule (10) to the commutativity of + from Section 3.1.3, with m as the induction variable gives:

$$\frac{\begin{array}{c}\forall l, m : \mathbb{N}.\ 0 + n = n + 0 \\ \forall m, n : \mathbb{N}.\ m + n = n + m \Longrightarrow s(m) + n = n + s(m)\end{array}}{\forall l, m, n : \mathbb{N}.\ m + n = n + m}$$

3.4.8 Targeted manipulation

The Targeted Manipulation sketch plan identifies a sequence of sub-terms of the current goal as *sources* that must be manipulated and a sequence of *targets* that help direct this manipulation. This need arises, for instance, during inductive proof when an induction hypothesis is used during the proof of the induction conclusion. In general, differences between a source and a target help locate the sub-terms to be manipulated and to measure success in this manipulation. Targeted manipulation is similar to rippling [4].

Example 10 **(Targeted Manipulation)**
Consider the proof of the step case of the associativity of +. The induction conclusion is the goal:

$$(l + m) + s(n) = l + (m + s(n))$$

where the left-hand side $(l + m) + s(n)$ is the initial source and the right-hand side $l + (m + s(n))$ is the initial target.

The left-hand side of the induction hypothesis becomes an intermediate target; this target is reached, from the initial source, by a simple application of the definition of $+$. This is followed by an application of the monotonicity rule for s applied to the induction hypothesis. The Targeted Manipulation conducted thus far is summarised as follows:

$$\begin{aligned}(l+m)+s(n) &= s((l+m)+n)Bydefinitionof+\\ &= s(l+(m+n))Byinductionhypothesis\end{aligned}$$

Targeted manipulation now compares the new source $s(l+(m+n))$ to the target $l+(m+s(n))$ to identify sub-terms that occur in the target, but not in the source, for instance, $m+s(n)$, which becomes a new target. An attempt is made to reach this next target, but the attempt fails. Instead, Targeted Manipulation now reasons backwards from this target to obtain another target.

$$\frac{m+s(n)=s(m+n)}{m+s(n)=x}$$

So $s(m+n)$ becomes an intermediate target, which guides the following manipulation:

$$\begin{aligned}\ldots &= s(l+(m+n))Previoussource\\ &= l+s(m+n)Bydefinitionof+\end{aligned}$$

The equation $m+s(n)=s(m+n)$ is now used in a similar fashion to the way in which the induction hypothesis was used; i.e., we apply a monotonicity rule for $+$ to the recursive definition of $+$ to obtain the final target.

$$\begin{aligned}\ldots &= l+s(m+n)Previoussource\\ &= l+(m+s(n))Bydefinitionof+\end{aligned}$$

The transitivity of $=$ is now used to equate the first and last terms in the chain of equalities and to conclude the proof.

3.5 User interface

MATHsAiD 2.0 is implemented in Amzi! Prolog. This version of Prolog includes an interface to Java, which is used to build a graphical user interface in Eclipse. A screen shot of this interface is given in Fig. 2.

Using this interface facilitates the rapid input of new theories and the generation of Theorems in these theories. LaTeX commands can be associated with symbols in the theory and are used by MathJax and JMathTex to render expressions in standard mathematical notation[15]. The interface is intended to be used by mathematicians without the need to understand the inner workings of MATHsAiD 2.0.

The user uses the 'Ops & Constants' tab to declare the operators and constants to be used in the theory, then provides definitions for them in the 'Axioms & Defs' tab. MATHsAiD 2.0 then automatically adds 'Theorems', 'Lemmas' and 'I-T Results'.

MATHsAiD 2.0 can be run entirely automatically or interactively, if the user wishes to guide its operations. The following interactive functionality is provided:

- Users may delete any automatically generated results they don't want, and they may form their own conjectures and ask the system to prove them. The user may choose to add any resulting theorems to the database as either lemmas or Theorems.
- In case it is unable to prove a conjecture, then users may ask MATHsAiD 2.0 to prove one or more intermediate lemmas, which they think may help it to prove the original conjecture.

3.6 Instructions for using MATHsAiD 2.0

New 'operators'(in the Prolog sense) need to be entered into MATHsAiD 2.0, before they be used in any rule (axiom/def/etc.). Each operator is defined in exactly one theory. Theories can be built on top of other sub-theories and, thereby, inherit all the operators and axioms in those sub-theories. It is only necessary for a user to load the uppermost theories. MATHsAiD 2.0 analyzes the operators used in the loaded theories and recursively loads those sub-theories in which these operators are defined. It is up to the user, not to re-define an operator they've already defined in a previous theory; and for that matter, not to introduce any inconsistencies. Figure 3 illustrates the introduction of a new operator.

Procedure 3 ***(Introducing a New Operator)***

The steps for introducing a new operator into a theory are as follows:

1. *In the main window (i.e., the one in* Fig. 2*), select a theory from the 'Theories' column on the left hand side (say 'Classes');*
2. *Click on the +Op button in the top centre of the main window. A new window appears, illustrated in* Fig. 3*;*
3. *Type the operator name in the top field;*
4. *Optionally, type in the Latex representation;*
5. *Set positioning and/or precedence, if different from default;*
6. *Optionally, preview the new operator;*
7. *Click OK.*

Note that the arity of the operator does not need to be specified. It will be inferred from any axioms, definitions or notations that use the operator.

Figure 4 illustrates the introduction of a rule, which could be a new axiom, definition or notation. Note that 'givens' refers to the premise of the rule.

[15] Realising the Standard Notation principle.

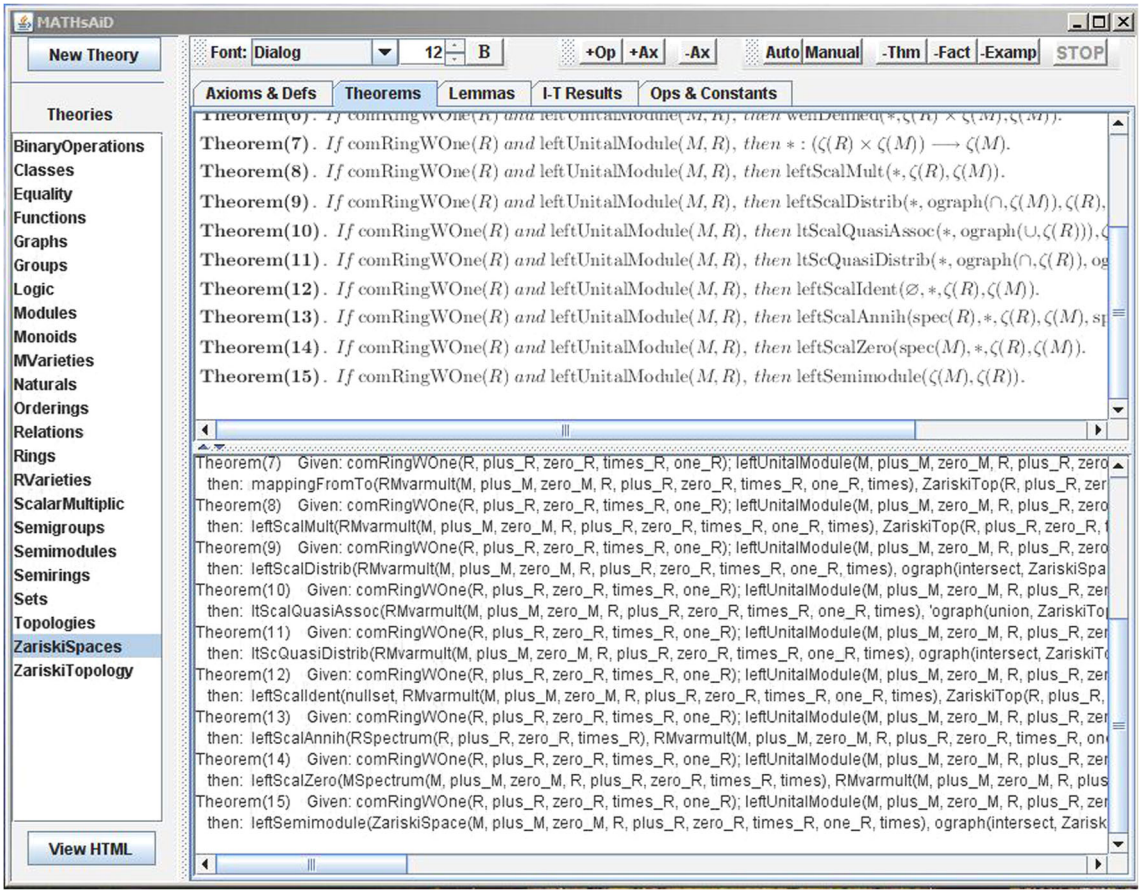

Fig. 2 The User Interface to MATHsAiD 2.0: The left hand tab allows the user to view different theories and add new ones. Within each theory, the other tabs allow the viewing and editing of: definitions, Theorems, lemmas, IT-results and operators. The bottom window displays the results in plain text and the top window displays the same results in more readable format using MathJax (http://www.mathjax.org/). Clicking on the 'View HTML' button displays the run of MATHsAiD 2.0 on the selected theory, rendered using JMathTex (http://jmathtex.sourceforge.net/). Displayed are the Theorems (including, as here, the inter-theory Theorems) from the Zariski spaces theory. In particular, MATHsAiD 2.0 has conjectured and proved the inter-theory Theorem that given an R-module M the Zariski space of M is a semi-module over the Zariski topology of R (see Section 3.3 for an explanation)

Procedure 4 (Introducing a New Rule)

The steps for adding a new rule are as follows:

1. *In the main window, select a theory (say Classes);*
2. *Click on the +Ax button (to the right of the +Op button). A new window appears, illustrated in* Fig. 4*;*
3. *Choose whether this is to be an axiom, definition or notation*[16]*;*
4. *Type in the rule name;*
5. *Input the givens (if any) and the conclusion, either by selection from the tables of previously defined operators and constants on the right hand side, or by typing them in.*
6. *Optionally, preview the new rule;*
7. *Click OK.*

[16]The only difference is in the appearance, which is in keeping with standard mathematical practice.

The table of 'Previously used givens', provided in the centre column, depend on the theory selected. Each time the user inputs a new rule (and clicks OK), its givens are added to the table for that theory. In most theories, givens, are often re-used, so providing them in this table reduces the amount of typing required, and the likelihood of errors. The tables of constants and operators are the same for all theories, since they do not tend to be so much associated with a particular theory.

Example 11 **(Introducing a New Rule)**

To define the new operator $symdiff$, it must first be introduced following the instructions in procedure 3, then a definition rule must be introduced. Suppose the givens are $class(A)$ and $class(B)$ and the conclusion is:

$$A\ symdiff\ B = (A \setminus B) \cup (B \setminus A).$$

The user could have first set the cursor in the Givens box, then clicked on $class(A)$, followed by $class(B)$. They

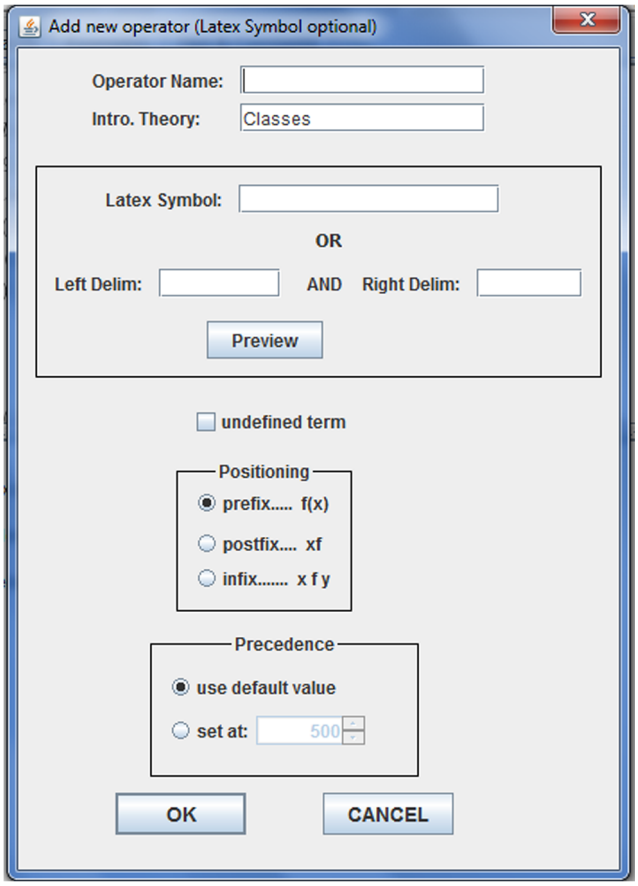

Fig. 3 Introducing a New Operator

could then have set the cursor in the Conclusion box and clicked on the following: $symdiff$, $=$, $\setminus$ and $\cup$. The required variables and parentheses would then need to be typed in manually.

4 Evaluation

The aim of the MATHsAiD 2.0 system, reproduced from Section 1, is:

> *To be a useful aid to the working mathematician, by conjecturing and proving many of the interesting Theorems of a given mathematical theory (from user-provided axioms), whilst limiting the number of non-interesting theorems generated.*

To be useful, MATHsAiD 2.0 should be capable of conjecturing and proving Theorems in theories of current mathematical interest, ideally including non-trivial Theorems. Such theories are usually high-level, in the sense defined in Section 3.3. It also requires an interface that is accessible to mathematicians who are not experts in automated reasoning. While we have presented such an interface in Fig. 2, we have not evaluated its usability in this paper.

Since theorem proving in non-trivial theories is undecidable, it is necessary to impose resource limits on MATHsAiD. These are primarily on the size of the conjectures generated, the lengths of their proofs and the time spent on working on them. These limits are under user control, but have default settings. The settings chosen for these limits can, not surprisingly, affect the results produced by MATHsAiD 2.0.

Note that in our previous work with MATHsAiD 1.0 (see Section 5.5), our primary goal was to demonstrate that an automated reasoning system could produce, at least in low-level mathematical theories, results comparable to those found in mathematics textbooks. That is to say, the system should be able to distinguish between interesting and non-interesting theorems. In [15] we provided some precision/recall data which supported our claim that this goal had been met. Because the current system, MATHsAiD 2.0, produces, in these same low-level theories, results quite similar to those produced by MATHsAiD 1.0, we do not include this data in our present evaluation.

Of course, there is no universal agreement on what constitutes an interesting Theorem, even for low-level theories; and certainly not for theories which are still actively being explored by mathematicians. Nor is it reasonable to expect perfect performance from MATHsAiD 2.0. It might omit to prove Theorems that some mathematicians might consider interesting and it might prove some theorems that they do not consider interesting. We must, therefore, temper any claim to allow for both disagreement on the 'gold standard' to be obtained and for minor deviations from perfection. With these caveats, the hypothesis to be evaluated in this section can be stated as:

> *MATHsAiD 2.0 can conjecture and prove interesting Theorems in high-level theories, including Theorems of current mathematical significance, without generating an unacceptable number of uninteresting theorems.*

4.1 Evaluation in the theory of zariski spaces

In order to evaluate this hypothesis, we chose Zariski spaces [17] to be our primary (high-level) theory in which we would determine whether MATHsAiD 2.0 could conjecture and prove any interesting Theorems. In particular, we wanted to see whether it could 'discover' some Theorems that have

[17] Briefly, the Zariski space of an R-module M (in this case, R is a commutative ring with 1 and M is unital) is the set of varieties of subsets of M, viewed as a semimodule over the semiring consisting of the Zariski topology of R. The variety of a subset A of M is the set of prime submodules of M which contain A.

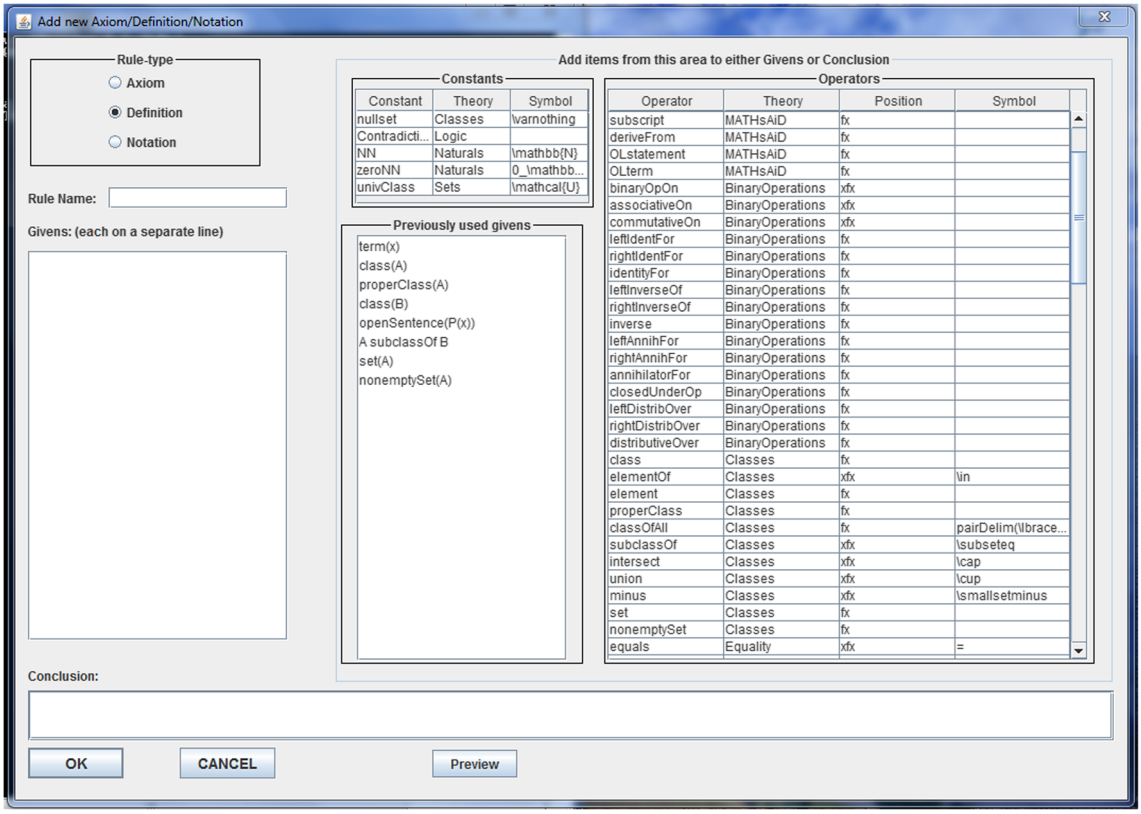

Fig. 4 Adding a New Rule

been published (recently) in a refereed mathematics journal. In contrast, by 'uninteresting theorems', we mean results which we deem not to be publishable. Lastly, 'an unacceptable number of uninteresting theorems' means a number sufficiently large as to either discourage one from looking, or in some way make it difficult for one to find the interesting Theorems in amongst the uninteresting ones.

In mathematical terms, the main motivation for considering Zariski spaces, and more generally, prime submodules, is that these concepts are generalisations of ring-theoretic constructs — constructs which are widely recognised as being of major significance in commutative ring theory. One would like to determine whether properties held by the ring-theoretic versions carry over to their module-theoretic counterparts. It turns out that, while a few important properties do indeed carry over, many do not. In fact, the module-theoretic concepts have proven to be far more complex (and some would argue, more interesting) than their ring versions.

From an automated reasoning perspective, the theory of Zariski spaces poses a real challenge; it is unusual for automated theorem provers to prove Theorems that relate multiple theories, let alone to conjecture such Theorems in the first place. As indicated in Section 3.3, Zariski spaces incorporate the theories of topology, (semi)modules, (semi)rings, (semi)groups, etc. That said, one does not necessarily have to develop a terribly large amount of module theory (for example) within a good automated reasoning system, in order for the system to reason about Zariski spaces.

All of the above are good reasons for choosing Zariski spaces. Add to these the fact that the first and third authors are two of the three original discoverers/inventors of this field of study[18].

In the event, MATHsAiD 2.0 did indeed conjecture and prove Theorems which have been published in refereed mathematics journals. In particular, within the theory of Zariski spaces, it 'discovered' a Theorem[19] that appears in [13]. As for whether MATHsAiD 2.0 also generated, along with these interesting Theorems, an unacceptable number of uninteresting ones, we were frankly surprised by how few uninteresting theorems were produced (even in the high-level theories); the number of interesting Theorems far surpasses the number of uninteresting ones. Table 2 gives some statistics on interesting *vs* uninteresting theorems for some sample theories. Further details can be found at http://dream.inf.ed.ac.uk/projects/mathsaid/currentResults.html.

[18] The other discoverer was M.E. Moore.

[19] 'Theorem (15)' in Fig. 2.

Table 2 Interesting *vs* Uninteresting Theorems: Note that only results that MATHsAiD 2.0 labelled as Theorems were classified

Theory	Interesting	Uninteresting
RVarieties	5	2
MVarieties	10	2
Zariski topology	4	1
Zariski spaces	13	2

The classification as 'interesting' or 'uninteresting' was made by the first author, who is an expert in these theories. Some of the 'interesting' Theorems were arguably only lemmas – albeit lemmas required in the proof of interesting Theorems

4.2 Failures of MATHsAiD 2.0

As discussed in Section 1, MATHsAiD 2.0 cannot be complete, so will sometimes fail. Its failures take two forms: interesting theorems it fails to discover or prove; and uninteresting theorems that it *does* discover, prove and mislabel as interesting. In this section, we give a few examples of each kind of failure.

4.2.1 Failure to prove interesting theorems

Here are some interesting theorems that MATHsAiD 2.0 did not discover in discovery mode, although it did prove some of them in theorem-proving mode, and could probably prove the rest given sufficient user investment in providing lemmas, etc.

We list first the theory and then the theorem name, followed by the theorem.

Logic: De Morgan's Laws:

$$\neg(p \vee q) \iff \neg(p) \wedge \neg(q)$$
$$\neg(p \wedge q) \iff \neg(p) \vee \neg(q)$$

Sets: One-way distributivity of powerset over union:

$$\mathcal{P}(A) \cup \mathcal{P}(B) \subseteq \mathcal{P}(A \cup B)$$

Functions: Function application is associative:

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Naturals: Associativity of multiplication:

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$

In many of these cases, MATHsAiD 2.0 *did* discover and prove a closely related theorem, but not the standard one. For instance, instead of the standard version of the associativity of multiplication, it found this commuted version:

$$(x \cdot y) \cdot z = (x \cdot z) \cdot y$$

which then made the standard version a trivial consequence and so uninteresting.

4.2.2 Failure by proving uninteresting theorems

Here are some uninteresting theorems that MATHsAiD 2.0 did discover and prove.

Relations:

$$\text{identRelatOn}(A) \subseteq A \times A$$
$$\text{totalRelat}(\text{identRelatOn}(A), A, A)$$

Naturals:

$$a + s(0) = s(a)$$
$$s(0) + b = s(b)$$
$$s(a) \cdot b = (a \cdot b) + b$$
$$a + (b \cdot c) = (c \cdot b) + a$$

These theorems are not totally uninteresting. Most of them are useful as Facts or Lemmas, because they enable the proofs of more interesting Theorems. They have just been mislabelled as Theorems.

5 Related work

By *mathematical theory exploration* we mean the generation of Theorems from the axioms of a mathematical theory. Since it is a relatively trivial matter to derive theorems by forwards reasoning from the axioms, the ultimate goal is to generate all and only the *interesting* Theorems. Such perfection is, however, both ill-defined and practically unobtainable. In well-developed theories we can define 'interesting' by appeal to what experts in the field have previously published as Theorems in research papers and textbooks, but even the experts will not be in perfect agreement. In new theories, which is where we hope MATHsAiD will find application, we can only appeal to the subjective opinions of the MATHsAiD users, referees and other observers. Even if we can agree on 'interestingness', it is only realistic to hope that an automated theory explorer will conjecture and prove *nearly* all and *nearly* only the interesting Theorems.

There are several other systems that automate mathematical theory exploration. We now briefly describe these systems and point out the principle differences between them and MATHsAiD 2.0.

5.1 Knuth-bendix completion

Completion, [10], is a technique for converting an arbitrary set of equations into a confluent set of rewrite rules, i.e., it defines unique normal forms. Coupled with the termination of the rewrite rule set, this provides a decision procedure for the theory defined by adopting the confluent set of rewrite rules as equational axioms.

The completion process works by finding a term ξ that can be rewritten into two terms ξ_1, ξ_2 with distinct normal forms $\hat{\xi_1}, \hat{\xi_2}$. Note that $\hat{\xi_1} = \hat{\xi_2}$, but they are syntactically distinct. The normal forms are put into their most general form ζ_1, ζ_2 and the result is called a *critical pair*. This critical pair can be oriented and added as a new rewrite rule, say $\zeta_1 \rightarrow \zeta_2$, so that $\hat{\xi_1}$ and $\hat{\xi_2}$ now *do* have a common normal form, namely $\hat{\xi_2}$. All rules in the set are put into normal form, so that some rules become trivial and can be discarded. The process is then repeated recursively. If it terminates, then it does so with a confluent set. It might, however, terminate with failure if, at some stage, no measure can be found to simultaneously orient the whole set. It might also not terminate, as it might be possible to continue to construct new critical pairs indefinitely.

Empirical results show that critical pairs often define interesting equational Theorems in their own right. When completion terminates, it often does so with an aesthetically pleasing set of equational axioms. Since these axioms also provide a decision procedure, it is not really necessary to develop the theory further.

Completion is attractive, but limited in its application. It works only for equational theories, although all theories can be encoded as equations. Clearly, it will not succeed on undecidable theories. It also has difficulty with inherently unorientable equations, such as commutativity, although these can sometimes be built into the unification algorithm instead. Attempts to apply it to inductive theories have so far proved unsuccessful (but see the discussion of IsaScheme in Section 5.3.2). MATHsAiD 2.0 is more general in that it also works in undecidable and inductive theories, and deals successfully with commutativity and other unorientable equations. It can also deal with non-equational reasoning in a natural way.

5.2 Proof planning

MATHsAiD 2.0's sketch plans are similar in spirit to *proof plans*, [3]. The idea behind both techniques is to capture common patterns of reasoning in mathematical proofs and use these to guide the search for new proofs. Most proof planning research has focused on inductive proof, with a particular emphasis on *rippling*: a plan for rewriting the inductive conclusion so that the induction hypothesis can be applied to it. When a proof plan fails, proof *critics* are used to analyse the cause of failure and to try to repair the proof, e.g., by conjecturing and proving a missing lemma, generalising the conjecture, using a different induction rule, etc.

Proof plans are less general, more focused and more prescriptive than MATHsAiD 2.0's sketch plans. For instance, rippling is aimed at a specific stage in inductive proof, specifies multiple steps of the proof and allows almost no branching. A typical sketch plan, on the other hand, applies to many proof stages, sometimes specifies only a single proof step and permits branching. Also, only proof plans utilise critics to repair failed proof attempts. Sketch plans instantiate the conjecture in parallel with proving it, whereas proof plans work with fully instantiated conjectures. Proof critics, however, like sketch plans, often work with conjectured lemma or generalisation schemas containing meta-variables that are instantiated as a side effect of their proof. In proof planning, this is called *middle-out reasoning*, because it allows instantiation choices to be postponed and determined retrospectively by later reasoning, i.e., the middle of a proof can be completed before the beginning is complete.

5.3 Inductive systems

5.3.1 IsaCoSy and hipster

IsaCoSy synthesises inductive consequences of recursive theories [7]. Recursive theories consist of recursive definitions of data-structures, such as natural numbers or lists, and recursive definitions of functions on these data-structures, such as addition, multiplication, append and reversal. The key idea underlying IsaCoSy is to generate only irreducible terms, i.e., terms in normal form with respect to a set of rewrite rules. These rewrite rules are formed by orienting all function definitions and previously proved Theorems, so that the rewrite rule set grows during theory exploration.

Requiring all conjectures to be irreducible is a surprisingly powerful interestingness heuristic.

- Firstly, all conjectures are simplified by being in normal form. This removes redundancy from their expression.
- Secondly, none of the conjectures can be proved by rewriting alone. In fact, no rewrite rules apply to them. Therefore, either induction or the backwards application of rewrite rules, is *required* to prove them, i.e., their proof is non-trivial.

Simple Theorems with non-trivial proofs tend to be interesting. This conclusion has been confirmed by a precision/recall comparison with the Theorems in the libraries of the Isabelle theorem prover. The idea behind this evaluation was that the Isabelle library Theorems have been manually chosen by Isabelle users as being interesting enough to be worth recording for the benefit of future users. IsaCoSy's precision was very good, i.e., it generated nearly all the Theorems in the Isabelle libraries. Its recall was not quite as good, i.e., it generated some theorems that were not in the library, but one could usually make a case that these extra theorems would have been reasonable additions to the library.

IsaCoSy uses a language of constraints to ensure that reducible terms are never generated. The left hand side of

each rewrite rule contributes constraints to ensure that no term is generated which it would match. Conjectures are equations between these irreducible terms. Conjectures are first sent to Isabelle's quickcheck counter-example finder [1] to filter out obvious non-theorems. Only a few conjectures survive this filter. These survivors are sent to the IsaPlanner proof planner [5] to be proved. IsaPlanner guides the Isabelle theorem prover [19] to find a proof. This whole process is completely automated.

The main difference between MATHsAiD 2.0 and IsaCoSy is that IsaCoSy was designed for purely-definitional, recursive theories, i.e., it usually has no non-definitional axioms, although there is nothing to stop a user adding such axioms. MATHsAiD 2.0 is designed to work with any kind of mathematical theory, including recursive theories, but is mainly aimed at algebraic theories, such as groups, rings, etc. Unlike IsaCoSy, MATHsAiD 2.0 does not use an irreducibility heuristic, but it does achieve similar effects by different mechanisms.

- By including simplification among its sketch plans, conjectures are put into a simplified form during their instantiation.
- By rejecting conjectures with a trivial proof sketch, MATHsAiD 2.0 ensures that its Theorems are non-trivial.

Another major difference is that MATHsAiD 2.0 simultaneously instantiates its Theorems from a conjecture shell and finds proof sketches for them. This ensures that it only generates Theorems and it does not need to filter its conjectures through a counter-example finder, which is the most time-consuming sub-process within IsaCoSy.

Hipster is a successor system to IsaCoSy [8]. It improves on IsaCoSy in the following respects:

- All functions in conjectures are translated only once into Haskell in order that they can be evaluated by Haskell's QuickCheck. Conjecture generation uses this Haskell representation of terms. In contrast, IsaCoSy uses Isabelle's representation of terms and applies Isabelle's QuickCheck to conjectures when they must be counter-example checked. Isabelle's QuickCheck translates each IsaCoSy's conjecture into ML, which means that functions are re-translated each time they appear in a conjecture, which is inefficient.
- Hipster uses Haskell's QuickCheck to evaluate each term on a selection of inputs. If terms agree on these inputs they are put into the same equivalence class. Conjectures are formed between a representative of each equivalence class and each other element in the class. This means that counter-example checking is not needed and the conjectures can be sent straight for proof in Isabelle.
- Although Hipster's success rate is comparable to IsaCoSy's (and to IsaScheme's, see Section 5.3.2), it is significantly more efficient.
- In Hipster, the user classifies tactics into routine or hard. Conjectures that can be proved using only routine reasoning are discarded as uninteresting. This is similar to MATHsAiD 2.0's use of trivial and generating proof plans.
- Like IsaCoSy, Hipster can generate interesting lemmas just from the recursive definitions of functions. Unlike IsaCoSy, it can also generate lemmas to unstick a stuck proof.
- Like MATHsAiD 2.0, Hipster is not restricted to inductive proof, but it has not yet been tested on non-inductive proofs.

The main differences between MATHsAiD 2.0 and Hipster are similar to those between MATHsAiD 2.0 and IsaCoSy.

5.3.2 IsaScheme

IsaScheme also synthesises Theorems [18], but using a different method. It uses a collection of user-determined schemes representing common forms of function definitions and conjectures to help ensure that the Theorems it generates are interesting. For instance, conjecture schemes might include associativity, commutativity, distributivity, idempotency, etc. Most of the evaluation has been conducted using a single, very general, conjecture scheme.

Of course, many instantiations of this scheme create non-theorems. As with IsaCoSy, the most obviously false conjectures are filtered out with a counter-example finder. IsaScheme also uses quickcheck, but additionally uses a second pass through the Nitpick counter-example finder [2]. The rationale is that quickcheck is quicker, but Nitpick finds counter-examples to more false conjectures. Survivors of counter-example finding are sent to a user-determined theorem prover to be proved, and those that are proved are candidate Theorems. Various Isabelle-based provers have been used for evaluation, including IsaPlanner, a custom-made, Isabelle induction tactic and Auto (for non-inductive theories).

If possible, the candidate Theorems of a theory are oriented as rewrite rules using a recursive path ordering [9]. Knuth-Bendix completion is applied to these rewrite rules in an attempt to turn them into a confluent set. If successful, the equations are extracted from this confluent set and are adopted as Theorems. It is not possible to orient all candidate Theorems, e.g., commutativity laws. These unorientable candidates are also adopted as Theorems.

Like MATHsAiD 2.0, IsaScheme has been applied to both inductive and non-inductive theories. Unlike MATHsAiD 2.0, IsaScheme may generate false conjectures, which

it filters out with counter-example finders. MATHsAiD 2.0 uses schemes, but only for a small class of Theorems. It also uses a termination order. This is currently a simple size ordering. Although this restriction has not yet proved problematic, it would be interesting to experiment with IsaSchema's more sophisticated recursive path ordering. It would also be interesting to explore the use of Knuth-Bendix completion.

5.4 Example-based theory exploration

Some theory-exploration systems are example-based, i.e., new concepts and conjectures are suggested by philosophical induction, e.g.,

$$\frac{P(0), P(1), P(2), \ldots}{\forall n \in \mathbb{N}.\ P(n)}$$

If proof is used at all, it is only to confirm these suggestions.

5.4.1 AM

AM generated a mathematical theory by using examples to suggest new objects, such as concepts, conjectures and examples, guided by a measure of interestingness that was inherited by new objects from those that led to its creation [12]. The AM system was composed by a collection of approximately 242 heuristic rules. Each rule was responsible for creating new objects from old, and assigning them an interestingness value. The creation of a new object would trigger further heuristic rules to be fired. These rules were placed on an agenda, ordered by their interest measure, so that those rules were fired first that were predicted to lead to the creation of the most interesting new objects. AM was initialised with 115 very general objects, such as sets, relations, etc. During a typical run, AM would generate of the order of 300 objects. These would include the natural numbers, prime numbers and arithmetic functions on numbers. During some runs, some important Theorems were suggested, such as De Morgan's Laws, the prime factorisation Theorem, Goldbach's conjecture, etc. After about 300 objects, a run would typically cease to generate interesting new concepts.

The main difference between MATHsAiD 2.0 and AM is the lack of proof, so that conjectures are only suggestive. In particular, MATHsAiD 2.0 formulates its conjectures by instantiating holes during proof. Moreover, it does not use a measure of interestingness to determine what is worthy of Theoremhood, and what just mere truth, but has a general set of criteria that a Theorem must meet.

5.4.2 HR

The HR system follows in the AM tradition, but:

- has only 10, very general, production rules for generating new objects;
- bases its interesting measure on few general principles, such as comprehensibility, parsimony, novelty and applicability;
- does have a proof capability, provided by the third party prover, Otter [16]; and
- uses another third party model generator, Mace [17] to generate examples of a concept.

The production rules operate on a common, table-based representation of the examples of a column. The production rules can speculate equalities between concepts, compose concepts, abstract concepts, etc. As a consequence of its generality and simplicity, HR has been successfully applied to a variety of domains, including finite algebra, number theory, and graph theory.

It has also been integrated with ideas on mathematical methodology due to [11] to correct faulty conjectures. For instance, given a faulty conjecture, HR can be used to learn concepts that distinguish those circumstances in which the conjecture is true from when it is false. The faulty conjecture can then be automatically repaired into a correct one [20].

The main difference between MATHsAiD 2.0 and HR is that MATHsAiD 2.0 discovers conjectures in parallel with its attempt to prove them, whereas HR only uses proof to confirm the correctness of conjectures induced from examples.

5.4.3 MCS

The Model-based Conjecture Searching (MCS) system, [21], uses a variety of third party theorem-proving and model-finding systems, such as Otter [16] and Mace [17] to generate and prove conjectures. Given an axiomatic theory, a set of finite models of these axioms are generated. Then a set of closed well-formed formulae are generated, consisting of equations, each of whose variables is either universally or existentially quantified. If a set of rewrite rules is provided then these formulae are rewritten into normal form. The models are used to classify these formulae into always true, always false and contingent. The always false ones are discarded and the always true ones become conjectures. An attempt is made to prove the conjectures automatically. Inductive learning is used to try to find relations between the contingent formulae, e.g., to find a minimal set of formulae whose conjunction implies another formula. Successful experiments have been conducted using various algebras, such as group theory, ring theory and quasi-group theory.

The main differences between MATHsAiD 2.0 and MCS are similar to those between MATHsAiD 2.0 and HR, namely: MCS's two stage conjecture and prove process, as opposed to the integration of testing into generation in MATHsAiD 2.0 and MCS's limitation to first-order, non-inductive theories.

5.5 The previous version of MATHsAiD

This paper describes MATHsAiD 2.0, which is a complete refactoring of the earlier MATHsAiD 1.0 [15]. The main differences between the old and new versions are as follows:

- MATHsAiD 1.0 only proved Theorems in low-level theories, such as set theory, whereas MATHsAiD 2.0 proves Theorems in high-level theories, i.e., ones, such as Zariski spaces, that are built on lower-level theories. These higher-level Theorems include some that have only recently been published.
- For instance, MATHsAiD 2.0 can prove inter-theory Theorems and results, i.e., Theorems about relationships between theories, e.g., that given an R-module M the set of all M-varieties forms a semimodule over the set of all R-varieties.
- In MATHsAiD 1.0 the logic was hard coded, but in MATHsAiD 2.0 it is user definable, provided it can be presented in a Natural Deduction format.
- MATHsAiD 2.0 uses both MathJax and JMathTeX to provide displays of mathematical symbols using LaTeX. The displays provided by MathJax are presented in the GUI itself; see Fig. 2 for an illustration. JMathTeX is used for rendering HTML files, one file for each theory. MATHsAiD 1.0 had no capability of displaying mathematical symbols.
- MATHsAiD 1.0 used only forwards reasoning, whereas MATHsAiD 2.0 uses both forwards and backwards reasoning, increasing its reasoning power.
- In MATHsAiD 1.0 users directed the system by a choice of axioms, whereas in MATHsAiD 2.0 the user selects only a collection of operators. We have found this to be a better match to user expectation.
- In MATHsAiD 1.0 candidate theorems were generated and then uninteresting ones were filtered out. The sketch plans in MATHsAiD 2.0 integrate these filters into the generation process, so that fewer uninteresting theorems are generated in the first place. This is more efficient. The new concept of *Theorem-producing rules* plays a key role in this process.
- MATHsAiD 1.0 used schemas for induction, whereas MATHsAiD 2.0 uses induction rules, making it easier to expand its inductive capabilities.

6 Conclusion

We have described the MATHsAiD 2.0 system, which is a tool for automated Theorem-discovery. Given an axiomatic theory, it automatically conjectures and proves Theorems of that theory. Our hypothesis is:

> *MATHsAiD 2.0 can conjecture and prove interesting Theorems in high-level theories, including Theorems of current mathematical significance, without generating an unacceptable number of uninteresting theorems.*

We have successfully evaluated this hypothesis by showing that MATHsAiD 2.0 is able to work in the theory of Zariski spaces, which is a topic which the first and third authors have worked on in their capacity as professional mathematicians [13]. In particular, MATHsAiD 2.0 was able to conjecture and prove a key Theorem from [13]. This Theorem appears as Theorem (15) in Fig. 2. This key Theorem is just one example of many inter-theory Theorems that MATHsAiD 2.0 has proved. It is unusual for automated theorem provers to prove Theorems that relate multiple theories, but they are a common aspect of modern mathematics, so it is essential for MATHsAiD 2.0 to demonstrate its abilities in this area.

We have attributed MATHsAiD 2.0's successful performance to its use of sketch plans and Theorem-producing rules. These two techniques combine to ensure that each Theorem has a short proof but does not have a trivial proof. Sketch plans only construct short proofs. If a trivial proof is found then the theorem is rejected or relabelled as a lemma. Theorem-producing rules ensure that a Theorem's proof ends with a non-trivial proof step. The absence of a trivial proof means that each Theorem adds some significant new information to the mathematical theory, so maximising MATHsAiD 2.0's precision. By constructing a series of short proofs it minimises the chances that an interesting Theorem is overlooked because it occurs only at an intermediate stage in a longer proof, so maximising MATHsAiD 2.0's recall.

The sketch plans interleave the construction of each Theorem with its proof. This ensures that only *interesting Theorems* are constructed. This obviates the need to filter conjectures with a counter-example finder to reject false conjectures, as done by IsaCoSy or IsaScheme, for instance. False conjectures are never constructed. It also obviates the need to improve interestingness by only generating conjectures in normal form, as done by IsaCoSy or MCS, for instance. Uninteresting theorems are rarely constructed.

Future work with MATHsAiD will focus on the following issues:

- Improving its range, so that it can produce interesting Theorems in more complex theories. This will require the development of additional sketch plans.
- Improving its usability, so that mathematicians can use it with very little preparation. For instance, we plan to offer it as a web service, so that it is not necessary for users to install it and they can use it via a simple graphical user interface. This will also make it possible for users to share theories via a central server, and hence easily add new theories on top of old ones.
- Backwards reasoning is currently guided by sketch plans. We will augment this guidance with additional search-control heuristics.
- We will improve the data-structures used for rule storage to enable more efficient look-up.
- We will focus search by preferring to build on the best Theorems. Theorems will be given an 'interestingness' weight which will then inform a best-first search strategy.
- We will explore mechanisms for repairing false conjectures, e.g., by speculating and adding preconditions under which they are true.

Acknowledgements The research reported in this paper was supported by EPSRC grants EP/F033559/1 and EP/J001058/1. Thanks to Grechuk Bogdan, Moa Johansson, Ursula Martin, Omar Montano Rivas, the audience of a Mathematical Reasoning Group seminar and an anonymous reviewer of our Applied Intelligence Journal submission for feedback on earlier drafts.

References

1. Berghofer S, Nipkow T (2004) Random testing in Isabelle/HOL. In: SEFM '04: Proceedings of the Software Engineering and Formal Methods, 2nd International Conference. IEEE Computer Society, Washington, DC, USA, pp 230–239. doi:10.1109/SEFM.2004.36
2. Blanchette J, Nipkow T (2010) Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann M, Paulson LC (eds) First International Conference on Interactive Theorem Proving (ITP 2010), vol LNCS 6172. Springer, pp 131–146
3. Bundy A (1991) A science of reasoning. In: Lassez JL, Plotkin G (eds) Computational Logic: Essays in Honor of Alan Robinson. MIT Press, pp 178–198
4. Bundy A, Basin D, Hutter D, Ireland A (2005) Rippling: Meta-level Guidance for Mathematical Reasoning, Cambridge Tracts in Theoretical Computer Science, vol 56, Cambridge University Press
5. Dixon L, Fleuriot JD (2003) IsaPlanner: A prototype proof planner in Isabelle. In: Proceedings of CADE'03, vol 2741. LNCS, pp 279–283
6. Hungerford T (1980) Algebra. no. 73 in graduate texts in mathematics, Springer-Verlag
7. Johansson M, Dixon L, Bundy A (2011) Conjecture synthesis for inductive theories. J Autom Reason 47:251–289
8. Johansson M, Rosén D, Smallbone N, Claessen K (2014) Hipster: Integrating theory exploration in a proof assistant. In: Proceedings of the conference on intelligent computer mathematics (CICM), vol LNCS 8543. Springer, pp 108–122
9. Jouannaud JP, Rubio A (1999) Logic in Computer Science, Symposium on The higher-order recursive path ordering, 0, 402. doi:10.1109/LICS.1999.782635
10. Knuth DE, Bendix PB (1970) Simple word problems in universal algebra. In: Leech J (ed) Computational problems in abstract algebra. Pergamon Press, pp 263–297
11. Lakatos I (1976) Proofs and refutations: The logic of mathematical discovery, Cambridge University Press
12. Lenat DB (1982) AM: An artificial intelligence approach to discovery in mathematics as heuristic search. In: Knowledge-based systems in Artificial Intelligence. McGraw Hill, New York. Also available from Stanford as TechReport AIM 286
13. McCasland R, Moore M, Smith P (1998) An introduction to Zariski Spaces over Zariski Topologies. Rocky Mountain J Math 28:1357–1369
14. McCasland RL, Bundy A, Autexier S (2007) Automated discovery of inductive theorems. In: Matuszewski R, Zalewska A (eds) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec, Studies in Logic, Grammar and Rhetoric, vol 10(23). University of Białystok, pp 135–149. http://mizar.org/trybulec65/
15. McCasland RL, Bundy A, Smith PF (2006) Ascertaining mathematical theorems. Electr Notes in Theor Comput Sci 151: 21–38
16. McCune W (1990) The Otter user's guide. Tech. Rep. ANL/90/9 Argonne National Laboratory
17. McCune W (1994) A Davis-Putnam program and its application to finite first-order model search. Tech. Rep. ANL/MCS-TM-194 Argonne National Laboratoriess
18. Montano-Rivas O, McCasland R, Dixon L, Bundy A Scheme-based synthesis of inductive theories. In: MICAI, LNCS, vol. 6437, pp. 348–361 (2010). Received best paper award
19. Paulson LC (1994) Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science **828**
20. Pease A (2007) A computational model of Lakatos-style reasoning. Ph.D. thesis, University of Edinburgh
21. Zhang J (1999) Jian: System description: MCS: Model-based Conjecture Searching. In: CADE-16: Proceedings Of the 16th international conference on automated deduction. Springer-Verlag, London, UK, pp 393-397